Toward compression-aware prefetching

Niloofar Charmchi, Caroline Collange

Compas'2019 : Parallélisme/ Architecture / Système
LIUPPA - IUT de Bayonne, France, du 24 au 28 juin 2019

Toward compression-aware prefetching

Niloofar Charmchi, Caroline Collange

Inria, Univ Rennes, CNRS, IRISA
firstname.lastname@inria.fr

Abstract
The speed gap between CPU and memory is impairing performance. Cache compression and hardware prefetching are two techniques that could confront this bottleneck by decreasing last level cache (LLC) misses. This paper explores opportunities for compression-aware prefetching, which creates a synergy between compressed cache and prefetching. The idea is to prefetch contiguous blocks, which can be compressed and compacted together, with the requested block on a miss access. Prefetched blocks that share storage with existing blocks do not need to evict a valid existing entry ; therefore, it avoids cache pollution. Based on our experimental evaluations, compression-aware prefetching has the potential to reduce the number of cache misses, by prefetching the potential blocks. We show that compression-aware prefetching could avoid up to 28% of misses by prefetching compressible and co-allocatable blocks in advance.

Keywords : Cache compression, Compaction, Hardware prefetching, Compression-aware prefetching

1. Introduction

One of the major challenges for computers is the speed gap between processor and main memory that affects the computer performance. In order to reduce this speed gap between off-chip memory and processor, on-chip memory, referred to as caches, is adopted. Caches store part of the working set and deliver those data to the processor quickly when it is needed. On-chip last level cache (LLC) is one of the most critical parts of a computer system. Since off-chip memory latency is high, finding techniques to minimize off-chip memory accesses is of essence. Methods such as cache compression and hardware prefetching can lead to a decrease in number of LLC misses [13, 7].
Hardware prefetching is an approach to reduce the number of memory accesses. Hardware prefetching is a speculative fetching of cache lines that have not been requested yet by the program. In this case, the prefetcher can diminish the cache misses. Multiple approaches have been proposed for hardware prefetching, such as stride prefetcher [4], next-line prefetcher [15], stream prefetcher [5] and offset prefetcher [7]. If the prefetches are performed accurately and early enough, they can hide off-chip accesses. Therefore, hardware prefetchers have become the center of attention among researchers in this field. However, there are some downsides in hardware prefetching. It can increase the contention for the available memory bandwidth, because apart from demand requests, prefetch requests also go to DRAM, which cause additional DRAM bank conflicts. Moreover, inaccurate prefetches increase the energy consumption

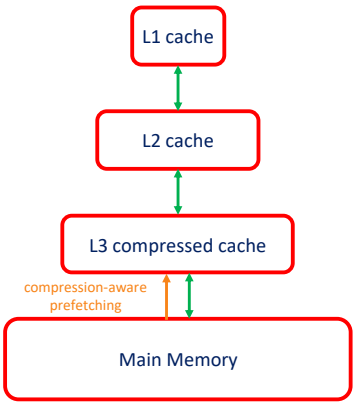


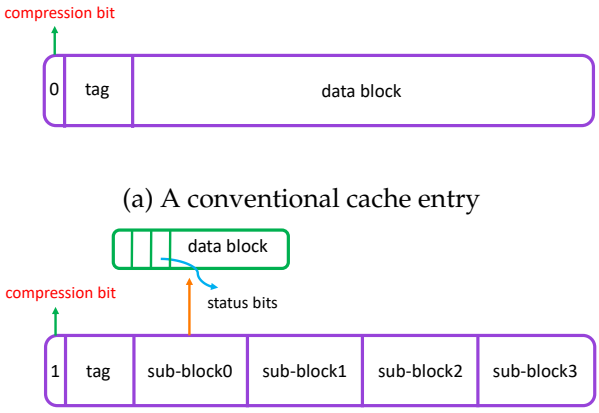
FIGURE 1 – Three-level cache hierarchy with compression-aware prefetching

because of unnecessary memory accesses, and prefetching can cause cache pollution if the prefetched data replaces useful data that will be needed later.
Cache compression is another important approach for performance enhancement in processors. The key idea of cache compression algorithm is to obtain the benefit of larger caches, while retaining the area and power of smaller caches. Compressed caches use compression techniques to reduce the size of the cache blocks and a compaction method to allocate compressed blocks together into one data entry. Therefore, cache compression schemes, such as Base-Delta-Immediate compression (BDI) [9] and Dictionary Sharing (DISH) [8], may lead to saving the off-chip traffic. Some well-known compaction techniques are Yet Another Compressed Cache (YACC) [13], Decoupled Compressed Cache (DCC) [14] and Skewed Compressed Cache (SCC) [12].

Our goal is to propose a compression-aware prefetcher, a unified system that benefits from prefetching and cache compression cooperatively. As it is shown in Figure 1, we have implemented compression in the LLC (L3 compressed cache). We evaluate the potential of compression-aware prefetching. Thus, by prefetching compressible blocks to L3, we seek the reduction in number of LLC misses. This positive interaction between hardware prefetching and cache compression would result in performance improvement. Therefore, prefetching and compression can take advantage of each other. Prefetching, however, increases the workload's working set size, it can evict useful data and causes misses. Compression can overcome this issue by increasing effective cache size. Thus, the extra capacity provided by compression can hide the disadvantage of prefetching.

2. Cache compression

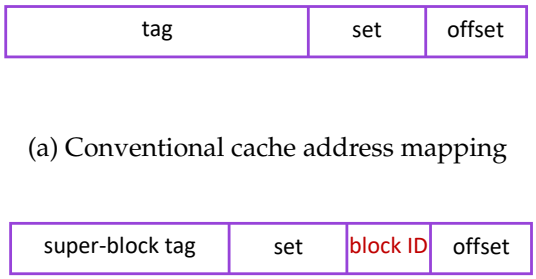
Unlike direct mapped caches, that each memory block maps to a single possible cache location, the n-way set-associative cache is divided into sets, and a block can map to one of the sets. Within a set there could be n blocks (n ways) ; therefore, each memory address maps to a specific set, but it can be allocated to any one of the n blocks in the set. The cache reads blocks from all ways in the selected set and checks the tags for a hit.
A sectored cache [6] can be designed using a shared tag among contiguous cache blocks. In this layout, the sectored cache is divided into a set of super-blocks and each super-block consists of



(a) A conventional cache entry

(b) A sectored/compressed cache entry

FIGURE 2 – Super-blocks in YACC can be conventional or sectored/compressed depending on the compression bit.



(a) Conventional cache address mapping

(b) Sectored/compressed cache address mapping

FIGURE 3 – Conventional cache and compressed cache address mapping

sub-blocks. Each super-block has a group of contiguous entries that share the same tag. Hence, using sectored cache reduces the tag area comparing to a conventional cache (Figure 2). Each sub-block of a super-block contains the data block and also some additional status bits such as valid and dirty bits.

The YACC architecture considers a compressed cache as a sectored cache with variable number of sub-blocks per super-block depending on the compression ratio. As it is shown in Figure 2, a super-block could have two layouts in a compressed cache based on the compression bit. If the super-block is uncompressed, it has the compression bit as zero and it contains only one sub-block. On the other hand, if the super-block is compressible, it can have multiple sub-blocks with the compression bit as one.

The address splitting of a conventional cache and a sectored/compressed cache are shown in Figure 3. These two caches have different hash functions and, therefore, they generate different access patterns as follows.

- In a conventional cache layout, the address is divided into offset, set and tag (Figure 3a).

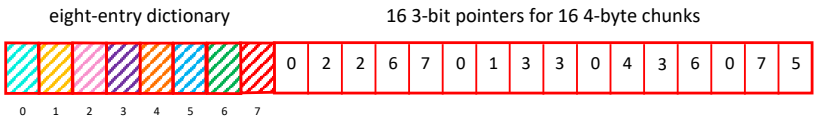


FIGURE 4 – A compressed cache block using DISH

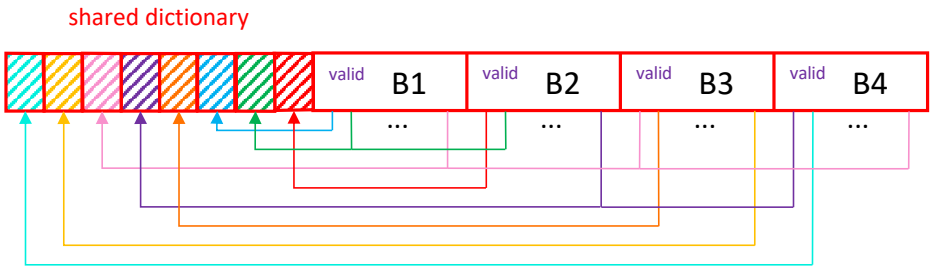
- In a compressed cache layout, how addresses are interpreted depends on the compression bit of the super-block (Figure 3b) :
  - In a compressed cache block address, block ID field indexes the proper sub-block of the super-block
  - In an uncompressed cache block address, block ID is used for comparison against the cache tags, together with the tag field. Thus, a compressed cache can hold multiple uncompressed super-blocks with the same tag in the same set, with different valid sub-blocks. We consider the extreme case of a compressed cache layout that has only uncompressed blocks as *baseline-modified* in section 3.2.

2.1. Dictionary Sharing algorithm

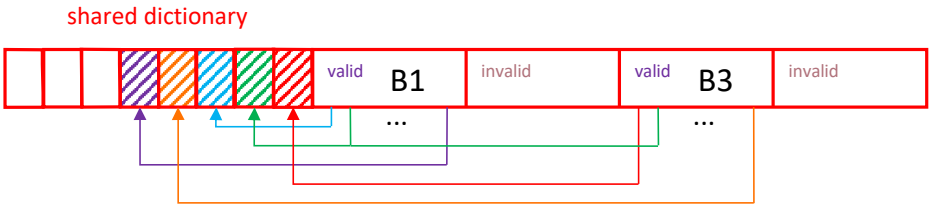
Dictionary Sharing [8] is a dictionary-based compression algorithm that shares a dictionary among all sub-blocks in a super-block. DISH treats a 64-byte cache block as 16 4-byte chunks. These chunks can be repeated in a cache block ; For example, a block may contain 0x00000000 in multiple chunks. DISH keeps each distinct chunk value in a cache block in an 8-entry dictionary, and replaces each of the 16 chunks with a 3-bit pointer to the corresponding dictionary entry (Figure 4). Each dictionary entry has a valid bit which is set when the entry contains a 4-byte chunk. If DISH does not find any invalid dictionary element to allocate a distinct 4-byte chunk, it considers the cache block as uncompressed. DISH also has another encoding scheme that is similar to the first one. It is possible that a block is compressible with both DISH schemes. In this case, a set-dueling [10] mechanism is applied to choose between the two schemes.
The key idea of DISH is to share a dictionary not only within a block, but also among multiple sub-blocks by taking advantage of the YACC layout. If the contiguous blocks are compressible with the same dictionary elements, they can be compacted into one super-block (Figure 5a). DISH compacts between one to four sub-blocks in a super-block. Suppose that block B1 is compressible with one of the DISH schemes (B1 is a valid sub-block in Figure 5a). When there is an access to block B2 (B1's neighbour block), if B2 is compressible with the same compression scheme and the same dictionary elements as B1, these two blocks can be compacted in the same super-block entry. Consequently, the cache entry contains two sets of pointers. The same can happen to other neighbour blocks of B1. DISH can compact up to four blocks in one super-block ; hence, it has maximum compression ratio of 4. Figure 5a shows a fully compressed super-block and the sub-blocks share the same dictionary. In Figure 5b there are only two valid sub-blocks that are co-allocatable.

3. Evaluating potentials for compression-aware prefetching

We have implemented DISH compression algorithm in the last level cache within the gem5 simulator [3]. Figure 6 shows a block diagram of compressed cache operations on an access.



(a) A super-block with four compressed sub-blocks



(b) A super-block with two compressed sub-blocks

FIGURE 5 – Co-allocation of multiple sub-blocks in a super-block using DISH

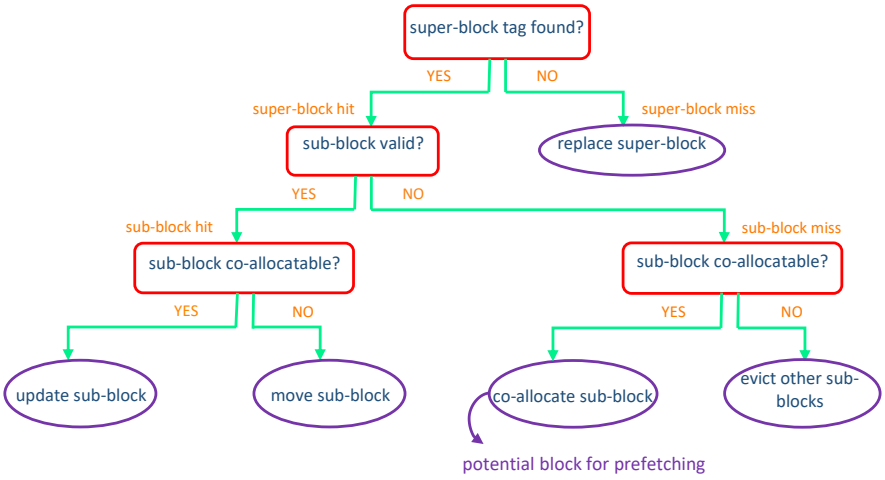


FIGURE 6 – Cache operations on write access

On cache write, DISH first looks for a matching super-block tag. If the tag is not found in the cache, it is a miss and the cache needs to replace the whole super-block ; otherwise, it is a super-block hit. Depending on sub-block validity, a sub-block miss or a sub-block hit could happen. In this work we focus on the co-allocation of a sub-block on a super-block hit and sub-block miss.
On cache read, in case of super-block miss or the sub-block is not present in the cache, it is a cache miss. However, if a super-block hit happens and the sub-block is valid and also compres-

TABLE 1 – Simulation parameters

Processor	4GHz, out-of-order
L1 cache	32kB, 4-way
L2 cache	256kB, 8-way
L3 cache	2MB, 16-way
Cache line size	64-byte
Replacement policy	LRU
Prefetcher at L2	Best-offset prefetcher

sed, the cache decompresses the sub-block.

3.1. Prefetching

Previous work [1] has introduced an adaptive prefetching scheme to detect useful prefetches using cache compression's extra tag. Furthermore, another studies [2, 11] have shown performance improvements in some benchmarks regarding to interactions between a compressed cache and simple prefetching; however, the disadvantage of prefetching is cache pollution. Thus, in this work we are looking for potentials for prefetching without polluting the cache and we want the prefetcher to be adapted to compressed caches.
Compression-aware prefetching technique targets sub-blocks that are co-allocatable and it prefetches them in advance. Based on Figure 6, potential prefetch targets happen on a super-block hit and sub-block miss. If the sub-block is compressible and co-allocatable in the super-block, we can count this sub-block as a potential for prefetching. In other words, if there is an access to an invalid block (e.g., B2 in Figure 5b), that is co-allocatable in the super-block, we could have prefetched it beforehand without any cache pollution. Thus, we could avoid this sub-block miss in the cache. In Figure 5b B2 is co-allocatable in the super-block if it can be compressed with the same dictionary elements as the super-block or it can add new entries to the empty slots of the dictionary. We will show that cache miss on a co-allocatable sub-block occurs frequently in cache accesses and prefetching potential blocks is advantageous on both reads and writes.
To evaluate DISH and its potentials for prefetching, we use SPEC CPU 2006 benchmarks. The statistics for the benchmarks are collected for 100M simulated instructions after warm-up of 50M instructions for 15 different execution snapshots. Table 1 shows the baseline configuration parameters with a conventional uncompressed cache.

3.2. LLC miss rate

We have simulated the compressed cache, which we implemented in LLC. Figure 7 illustrates the reduction in number of LLC misses for 28 benchmarks. We consider the conventional cache (baseline), DISH and three other configurations :

- 2Xbaseline : a sectored cache with two sub-blocks in each super-block. It has a capacity of 4MB in LLC.
- 4Xbaseline : a sectored cache with four sub-blocks per super-block. The LLC size of this configuration is 8MB.
- baseline-modified : another configuration of 2MB conventional cache with the same address mapping and hash functions as the compressed cache (Figure 3b).

The results of baseline-modified, DISH, 2Xbaseline and 4Xbaseline caches normalized to a conventional uncompressed cache are shown in Figure 7. Even though a 2MB compressed cache can approach a 4MB conventional cache (2Xbaseline) in some of the benchmarks, in total

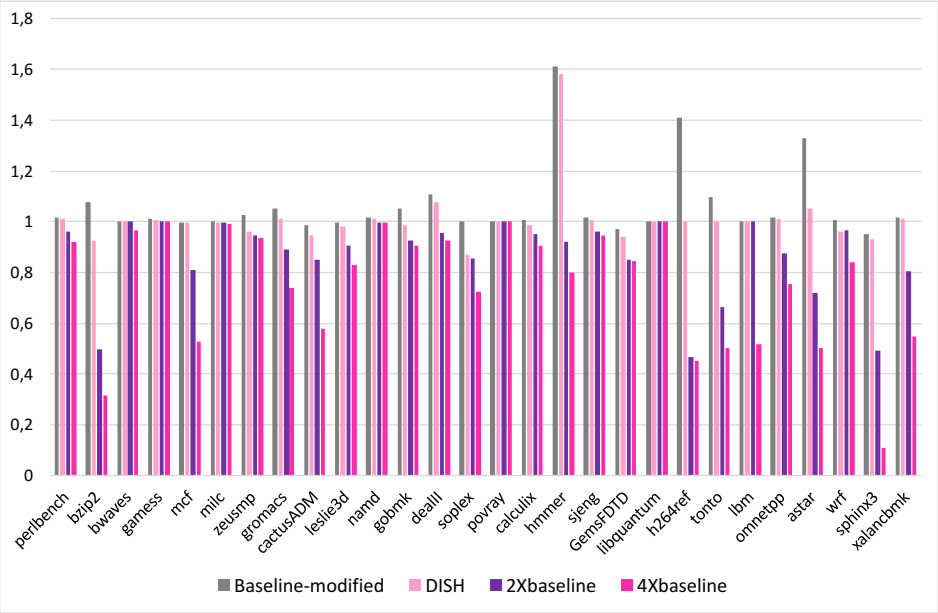


FIGURE 7 – Number of LLC misses normalized to a 2MB uncompressed cache

cache compression provides modest improvements, or even an increase in cache misses due to the change in address-to-set mapping.

3.3. Opportunities for compression-aware prefetching

In order to evaluate the potential of compression-aware prefetching, we count the prefetching opportunities. As mentioned in section 3.1, the misses that occur on a super-block hit and sub-block miss are avoidable if they are co-allocatable in the super-block (referred to as potential block for prefetching in Figure 6). We calculate the percentage of misses that could be avoided by taking advantage of compression-aware prefetching. Figure 8 shows the reduction in number of misses in terms of MPKI in case of prefetching potential blocks. We could reduce the MPKI up to 28%, by prefetching compressed blocks before they are requested by the processor.

4. Conclusion

This paper evaluates significant potentials of prefetching by proposing a special hardware prefetcher that is adapted to compression (using DISH compression algorithm in LLC). This study has shown that there is a potential for a synergistic interaction of prefetching and cache compression in processor architecture. As future work, we will design an actual compression-aware prefetcher. In order to implement the prefetcher, we need a predictor that predicts whether other sub-blocks will be used and co-allocatable in the same super-block. By applying a compression-aware prefetcher on miss accesses, we can prefetch the contiguous blocks if the block is predicted to be compressible. In order to achieve better results on compression-aware prefetching, we will apply compression on two cache levels (L2 and L3) and prefetch blocks to

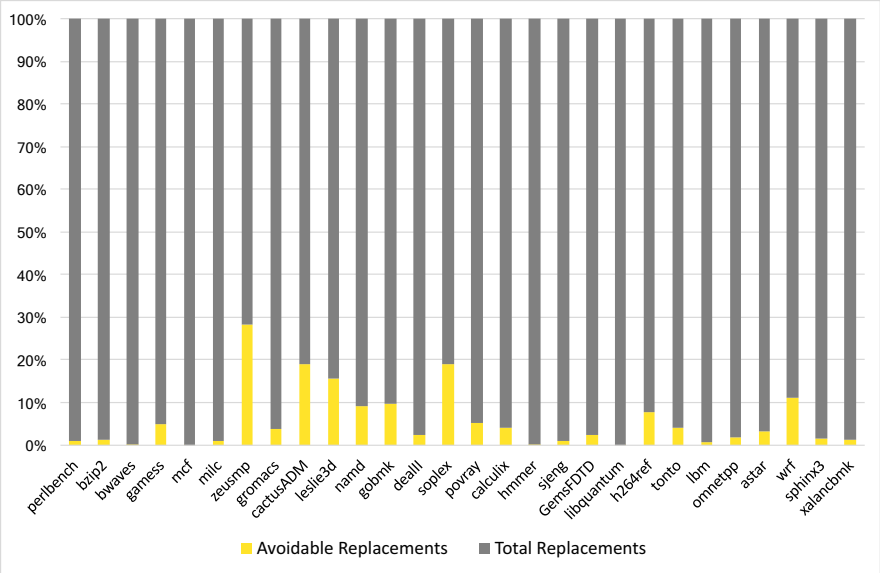


FIGURE 8 – Percentage of avoidable misses with compression-aware prefetching

L2.

5. Acknowledgements

Thanks to André Seznec and Daniel R. Carvalho for helpful comments and suggestions.

Bibliography

1. Alameldeen (A. R.) et Wood (D. A.). – Interactions between compression and prefetching in chip multiprocessors. – In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 228–239. Ieee, 2007.
2. Arelakis (A.) et Stenstrom (P.). – Sc 2 : A statistical compression cache scheme. – In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pp. 145–156. IEEE, 2014.
3. Binkert (N.), Beckmann (B.), Black (G.), Reinhardt (S. K.), Saidi (A.), Basu (A.), Hestness (J.), Hower (D. R.), Krishna (T.), Sardashti (S.) et al. – The gem5 simulator. *ACM SIGARCH Computer Architecture News*, vol. 39, n2, 2011, pp. 1–7.
4. Fu (J. W.), Patel (J. H.) et Janssens (B. L.). – Stride directed prefetching in scalar processors. *ACM SIGMICRO Newsletter*, vol. 23, n1-2, 1992, pp. 102–110.
5. Hur (I.) et Lin (C.). – Memory prefetching using adaptive stream detection. – In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pp. 397–408. IEEE, 2006.
6. Liptay (J. S.). – Structural aspects of the system/360 model 85, ii : The cache. *IBM Systems Journal*, vol. 7, n1, 1968, pp. 15–21.

7. Michaud (P.). – Best-offset hardware prefetching. – In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 469–480. IEEE, 2016.
8. Panda (B.) et Seznec (A.). – Dictionary sharing : An efficient cache compression scheme for compressed caches. – In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12. IEEE, 2016.
9. Pekhimenko (G.), Seshadri (V.), Mutlu (O.), Gibbons (P. B.), Kozuch (M. A.) et Mowry (T. C.). – Base-delta-immediate compression : practical data compression for on-chip caches. – In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pp. 377–388. ACM, 2012.
10. Qureshi (M. K.), Jaleel (A.), Patt (Y. N.), Steely (S. C.) et Emer (J.). – Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, vol. 35, n2, 2007, pp. 381–391.
11. Raghavendra (K.), Panda (B.) et Mutyam (M.). – Pbc : Prefetched blocks compaction. *IEEE Transactions on Computers*, vol. 65, n8, 2016, pp. 2534–2547.
12. Sardashti (S.), Seznec (A.) et Wood (D. A.). – Skewed compressed caches. – In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 331–342. IEEE Computer Society, 2014.
13. Sardashti (S.), Seznec (A.) et Wood (D. A.). – Yet another compressed cache : A low-cost yet effective compressed cache. *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, n3, 2016, p. 27.
14. Sardashti (S.) et Wood (D. A.). – Decoupled compressed cache : Exploiting spatial locality for energy-optimized compressed caching. – In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 62–73. ACM, 2013.
15. Smith (A. J.). – Cache memories. *ACM Computing Surveys (CSUR)*, vol. 14, n3, 1982, pp. 473–530.